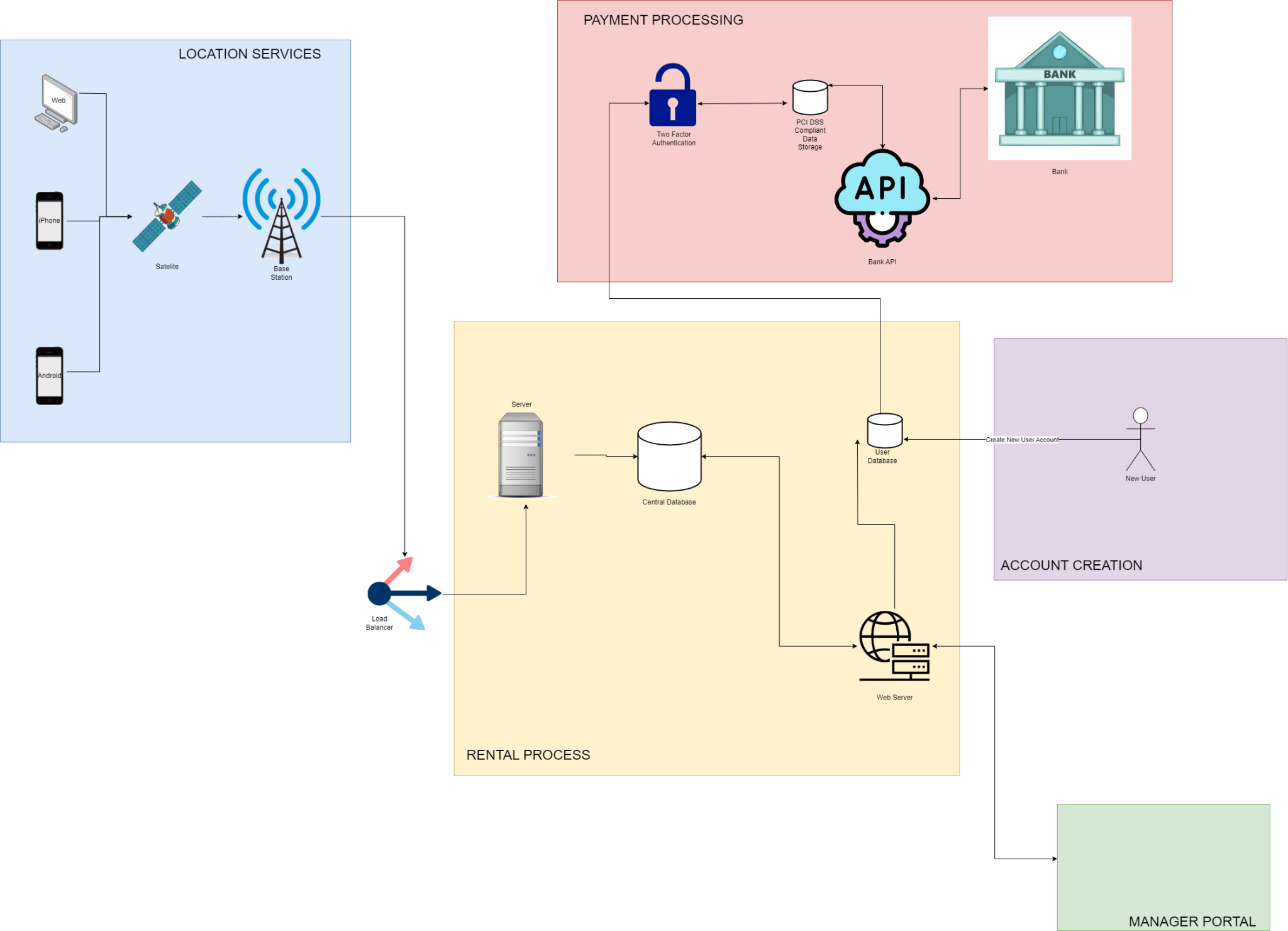
BeAvis Car Rental System Design Specifications

Team Members: Abdihakim Ahmed, Jake Foster, Alex Huang

**System Description**

The BeAvis Car Rental System is designed to replace the company’s outdated rental process and should be accessible as an app on a mobile phone or as a website. Whether on the app or website, users must first make an account and verify their identity by correctly entering the verification code sent to them. Once the user sets up their account, they can make a rental, look at locations, and view the rental history. Users are encouraged to pay through the app or website to ensure a quick and secure transaction. The payment information is stored separately from other information to minimize the fallout of a potential privacy breach. This document will provide an optimal software architecture diagram as well as the best UML diagram so that the BeAvis software system can be correctly designed and implemented.

**Software Architecture Diagram**

****

**SWA Description**

Our software architecture diagram follows the flow from left to right and starts with the three possible uses whether it is on an iPhone, Android, or the web. The box labeled “Location Services” on the left is meant to satisfy the requirement of allowing the user to enable their location which would give them access to locations near them. First, satellites communicate with a device's GPS to pinpoint the location of it. This location is then sent to the base station and the user is now able to turn on their location. Before reaching the server, we have a load balancer so that the server maintains efficiency and speed.

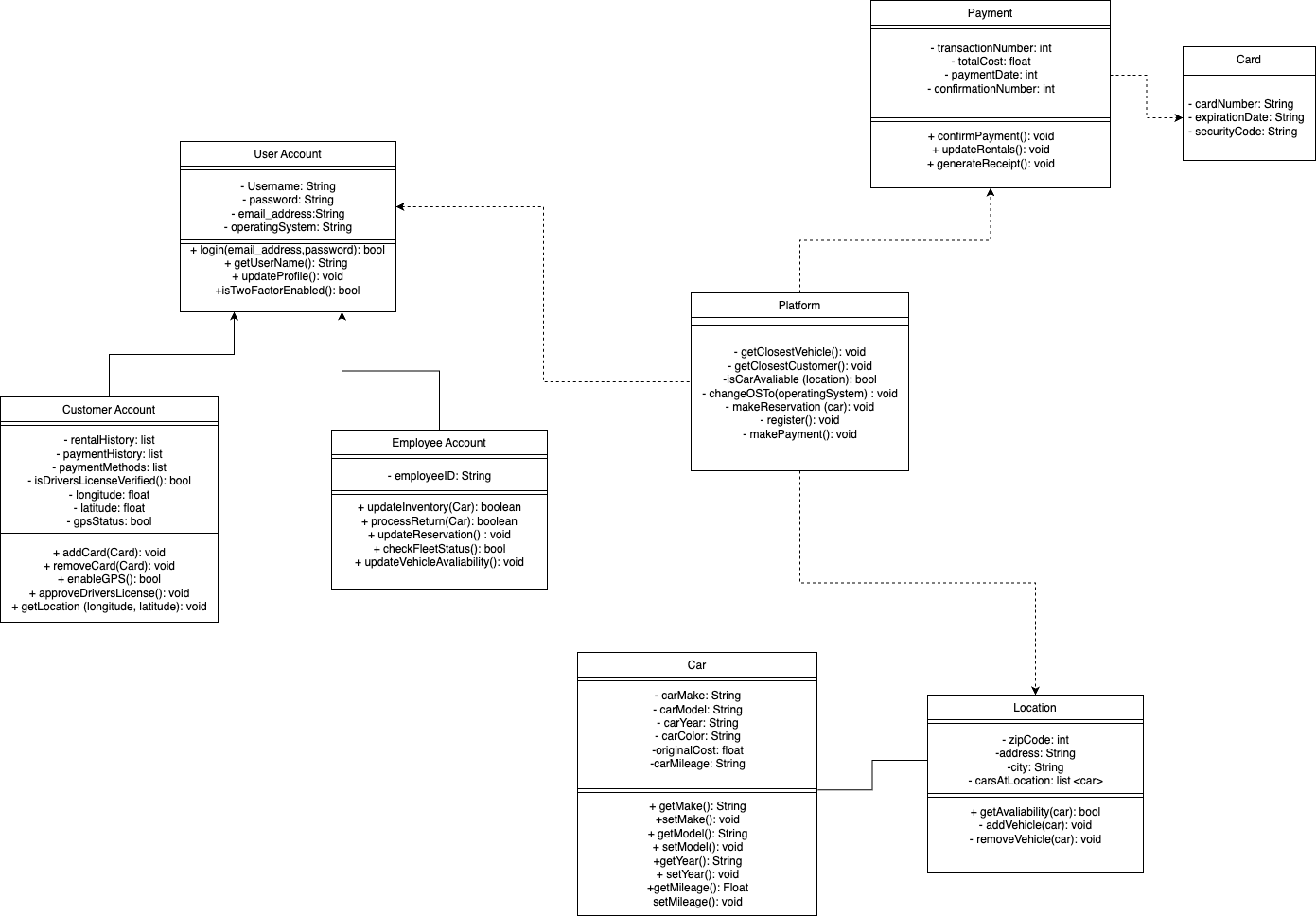
In the “Rental Process” of the diagram, the user finally can connect to the server. Once the connection is established, the server accesses all of the information within the central database which stores all of the rental information. Next, we must implement access to a web server so that the user can make rentals based on updated information within the system. New rentals must be taken into consideration and sent back to the central database. Since the central database and web server are communicating back and forth, we used a double-sided arrow. The web server then accesses the user database which stores important details about the users account such as their rental history or account information.

For account creation, we simply just connect to the user database to make and store the user's password, email, and other account information. Nothing else is necessary for account creation. This also doesn’t follow the traditional left-to-right flow of our diagram as creating an account is very simple which allows us to bypass mostly everything else in the diagram.

The Manager Portal communicates back and forth (double-sided arrow) with the web server as they would be able to make adjustments or updates to the available rentals as well as view customers’ rental history. The manager portal would communicate this information with the web server which would then be sent back to the central database.

In the “Payment Processing” section, there is a two-factor authentication that the user must complete before making a payment. This is only necessary for making a payment and is not required to view the information on the site. Then the system accesses a payment card industry data security standard database to successfully and safely deal with purchases. However, if the user is not verified, they will be sent back to authenticate themselves which is shown with the double-sided arrows. Next, a bank API is accessed so that the payment process can be super quick and easy for the user. Finally, the bank API sends the payment information directly to the bank so that the sender's bank can send the correct amount of money to the receiver's bank. We modeled the payment section of our software architecture diagram after that of Venmo ([How To Make An App Like Venmo - IdeaUsher](https://ideausher.com/blog/venmo-like-app-development/)).

**UML Class Diagram**

****

**Diagram Description**

**Platform Class:** The platform class is the center of the UML diagram as it deals with various functions with parameters from different classes. Without it, our system would not function properly. It consists of a database that uses different types of methods such as getClosestVehicle() and getClosestCustomer(). This would assist a customer with choosing a vehicle closest to the customer based on the location that is inputted. isCarAvaliable(location) is the method that inserts the availability of a car that is contingent to the location. The return type for this method is a boolean, contingent on whether the car is available or not.

changeOSTo(operatingSystem) gives the option to the customer to change what operating system that can be used whether it is an iPhone, Android, or a web browser. makeReservation(car) is a method that creates a reservation to obtain a car. Then there is the register() method which constructs and registers a user account into the platform/database. Lastly, makePayment() initiates the payment procedure when completing the reservation.

This class depends on many of our other classes to carry out these methods successfully. The Platform class must be connected to the Payment class as it would need to access the information within that class in order to use makePayment(). The platform class also depends on the Location class because methods such as isCarAvailable() require the parameter of location. Finally, the Platform Class depends on the User Account class and this is seen where attributes from the User Account class such as operatingSystem are passed through into methods within the Platform class. Therefore, the Platform class is dependent upon the Payment class, Platform class, and User Account class.

**Card Class:** We designed a card class that holds the information of a customer’s credit card such as the credit card number, security code, and expiration date. This information should all be stored in the form of a string. This class has no operations as we simply want to gather information to instantiate a card object.

**Payment Class:** There is a payment class that has the attributes of the cost of the rental, the date of payment, the transaction number, and the confirmation number. All of these should be integers except for the cost which would be a float. The class also has methods such as confirming that a payment went through and generating a receipt for a given purchase. Once a purchase is made, the operation of update rentals may be accessed to acknowledge that one of the vehicles was recently rented out. None of these operations require a return of any sort so we set them all to void. We also used a dependency relationship between payment and card as the payment class requires credit card information to be successful.

**Car Class:** The car class holds vital information about all of the rental cars within the system such as their color, make, model, year, cost, original cost, and mileage. However, the information is general and not relative to their location. Original cost should be in the form of a float and all other attributes should be in the form of a string. The class also has getters and setters to access and establish the most important information of the cars such as the make, model, and year. These simple operations allow for easy updates and retrieval of data.

**Location Class:** The location class has the attributes of zip code, address, city, and carsAtLocation. Address and city are both strings, zip code is an int, and carsAtLocation is a list that passes in information from the car's class. Essentially, the carsAtLocation attribute would hold all of the available rentals per specific location so instead of displaying every single car possible it would show only the ones at that place. For operations, we have getAvailability(car) and take in the parameter of car. This method checks to see if a specific car is at a given location. This method will return true if it is found and false if not. Next, we have addVehicle(car) where a car is passed in as well. This would be used to add a specific vehicle to a specific location and also has a return type of boolean. Finally, we have removeVehicle(car) which removes a given car from a location. We wanted to make sure that these updates were done within the location class as different locations may have different vehicles available. There is an association between this and the car class as each location has various amounts of cars.

**User Account:** The user account class takes in strings of the users’ email, password, username, and operating system. Key operations such as login exist which take in the user's email and password and return a boolean depending on if it was successful or not. This class also has a method to check whether the user has two-factor authentication enabled and returns true or false. To add, we have simple methods such as getUsername() which simply retrieves the username or updateProfile() which would allow the user to update or change their account information. The User Account class also acts as the parent class for the Customer Account and Employee Account.

**Employee Account:** This class only requires the employee's ID, as it inherits the rest of the information from the user account class. We drew a generalization connection between the employee account and user account classes as they are both accounts. However, this class has additional methods that only an employee can use, such as updateInventory(car) and processReturn(car). After passing in the correct parameter, employees can update their inventory based on recent purchases or even deal with returns. These two operations return true or false depending on if they were successful. Next, we have methods to update a reservation or update the availability of a vehicle. These two have a void return type as they are simply tasks. Finally, we have an operation to check the status of all of the cars at a location that has a boolean return type. This may provide useful information through a general report of all of the cars. These are all very important features that employees are encouraged to use to do their jobs.

**Customer Account:** Similar to the employee account class, Customer Account inherits from user account so there is a generalization arrow connecting the two. A customer account is just one form of a user account. Customer Account is where customers can view important things such as their rental history, payment history, and payment methods. This information allows customers to be able to view their past business with the rental company and should be stored in lists. Next, there is an attribute to check if the customer has a valid driver's license with a boolean return type. The final 3 attributes in this class all deal with the customer being able to enable their location to see nearby rental places. We have longitude and latitude which are given as floats, and a separate attribute that tells us whether the GPS is turned on or not. For operations, we have methods to add a credit card and remove a credit card. For these, we pass in “card” and return void. There is also an operation with a boolean return type that allows the customer to enable the GPS so that they can view nearby locations. We also added an operation that returns nothing to approve the customer’s driver's license and verify it. Lastly, the getLocation method takes in the customer’s latitude and longitude, or current position, and grabs their location. This doesn't need to return anything.

**Development Plan and Timeline**

Step 1: Planning and Brainstorming, assigning roles and responsibilities to team members

This step should take about 2-3 weeks. It is critical to our operation because we will be defining the fundamentals of the application, and will work as our framework, a.k.a. guide going forward. This will include everything from application requirements to defining core functionalities of the car rental system such as booking and payment. Here, we will also assign responsibilities to the team based on their skill sets.

Every team needs a leader, we will elect a project manager to make difficult decisions and coordinate with everyone. Since this is an application for commercial use, there will be a frontend and backend part. To save costs, we will assign one team member to work on the front end, and another team member to work on the backend. The frontend team will be designing the app interface to improve user experience, while the backend developer will be building databases, and writing logic to make the code work. Since the backend is entirely dedicated to one team member, we will hire a quality assurance team to make sure that our product is safe for commercial use. This means they will be testing for bugs and confirming that the functionalities align with our project specifications. Finally, to make sure our financials and balance sheets are cash-flowing, we will hire an economics/marketing professional to help us reach out to more customers and optimize our overall performance based on customer feedback.

Step 2: Focusing on the user interface

Having a good user interface and an overall idea of how to organize where key components are located, will help us establish a good overview on how to tackle the backend challenges. Here, we will be dedicating about 3-4 weeks to design the user interface of the application. We will be creating roadmaps of the different paths the user could take to reach a desired destination, optimizing as we get along. The UI designer will be figuring out the best color scheme, font, icons, etc.

Step 3: Backend Development

With a good user interface, the backend developer will have a clear understanding of how they should be implementing the features. Since core functionalities are developed by the backend, we will allocate at least 8 weeks to get it completed. Here, we will need a robust database to safely secure user information and meet industry standards for security. We will be implementing the UML diagram classes and functionalities, such as vehicle inventory, reservations, and payments. Furthermore, these implementations will require us to build APIs for easy access and to be able to link our backend with our front end. Backend development typically results in the most bugs due to logic errors; therefore, we will be allocated 1-2 weeks to write out unit tests to have 100% code coverage, making sure it also integrates fully with our frontend components.

Step 4: Frontend Development

In step 2, we designed the overall look and feel of the application but did not get in-depth about how we were going to implement it. In step 4 of the development process, we will be connecting the front-end interfaces to backend APIs. This means developing UI components working with backend functionalities to enhance user experience. Examples include search filters and optimized form filling. Furthermore, at this stage of the development process, we want to make sure performance is fast and responsive. This process will take about 4-6 weeks.

Step 5: Testing and Quality Assurance

We have functional requirements specified in our specification requirements, in this phase, we will be validating each feature to make sure that it is working as intended and identify bugs. Furthermore, we will be hiring anonymous users to test the app and provide feedback on its usability and experience. We will also be hiring penetration testers to perform security tests to ensure user data is properly encrypted and safely stored in our databases, especially for our payment functionalities. This process will take about 2-4 weeks.

Step 6: Launch and Maintenance

**We did it!** The app is ready to be released. To ensure last-minute quality control, we will be releasing the app to a small group of people to identify last-minute issues. Once that successfully passes, we will deploy the app to production and our marketing team will be in action, optimizing the app based on the feedback we get from customers. This process shouldn’t take more than 1 week. However, we are mindful there are issues that we might have not thought of and maintenance will be a common thing that has no deadline and will be ongoing. In this phase, we will be monitoring user feedback, fixing bugs, and rolling out new features based on user feedback.

**Sources:** Grammarly was used to proofread my work